Lab: Constructors and Operator Overloading

Learning Outcomes

- Gain practice in class design by implementing a complete class
- Gain practice implementing constructors
- Gain practice in overloading operators

Overview of Classes

In the previous laboratory you used opaque pointers to create a singly-linked list abstract data type (ADT). Recall an ADT is the description of a set of data items and of the operations that can be performed on that data. The *abstract* part means that the data type is described independent of any implementation details. Opaque pointers are a way to hide the implementation details of an interface from ordinary clients, so that the implementation may be changed without the need to recompile the source files using it. C is not known for its support of implementing ADTs. However, you used a C technique to encapsulate data using an opaque pointer and to support a certain level of polymorphic behavior. By hiding a data structure's implementation and its supporting functions, the client doesn't need to know how the structure is implemented. Hiding this information will reduce what the client needs to know and thus reduce the application's complexity level. In addition, the client will not be tempted to take advantage of the structure's internal details, potentially causing later problems if the data structure's implementation changes. As discussed in lectures, this type of data abstraction and encapsulation can be implemented with classes with the added advantage of automatic execution of constructors and destructors.

In this lab, you will apply many important C++ concepts discussed during this week's class lectures to implement your first C++ ADT using classes. Let's recap these concepts:

- ***Constructors*** are member functions that are automatically invoked to initialize the state of objects when they are created. If a class has defined an appropriate constructor, it will guarantee that all objects of that class are initialized before the programmer can do anything else with the object, thus ensuring that the programmer will always be working with properly initialized data. Constructors can be overloaded provided each function has a unique signature or combination of number or parameter types. Constructors are a complex topic with a variety of constructor types [default constructors, copy constructor, move constructors, single-argument conversion constructors, delegating constructors], and additional concepts involving member initialization lists and keywords `explicit` and `default`. To provide safe behavior, every C++ class should have one or more constructors.
- ***Destructor*** is a member function that automatically runs when an object is destroyed. The destructor is responsible for whatever operations the class designer wishes to have executed subsequent to the last use of an object. Typically, the destructor frees resources an object allocated during its lifetime. Object destruction happens implicitly when the object goes out of scope, or as a result of the programmer explicitly deleting an object.
- ***Dynamic memory*** is managed through a pair of operators: `new`, which allocates, and optionally initializes, an object in dynamic memory and returns a pointer to that object, and `delete`, which takes a pointer to a dynamic object, destroys that object, and frees the associated memory. Operators `new[]` and `delete[]` are the counterparts of operators `new` and `delete`, respectively for allocating and deallocating arrays. Dynamic memory is

problematic because it is surprisingly hard to ensure that memory is freed at the right time. Any class that performs dynamic memory allocation in a constructor will typically need a copy constructor, an overload assignment operator, and a destructor.
- ***Access specifiers*** `public`, `private`, and `protected` provide protection for class data members and member functions from access outside the class by declaring them in public, private, and protected sections of the class, respectively. Anything declared in the public section is freely accessible from anywhere outside the class. Items in the private section are accessible only to member functions within that class. Member functions and data in the protected section are treated as private, except that classes that are derived from the given class are given access to anything in the protected section. Data abstraction and encapsulation techniques require only interface functions be placed in the public section while implementation and data representation details be placed in the private section.
- `this` is an extra, implicit parameter that is passed to member functions to access the object on which they were called. When a member function is called, `this` is initialized with the address of the object on which the function was invoked.
- `const` ***member functions*** cannot change the object on which they are called. A function indicates `this` is a pointer to `const` by putting `const` after the parameter list of a member function. Objects that are `const`, and references or pointers to `const` objects, may call only `const` member functions.
- ***Operator overloading*** allows programmers to define what operators mean when applied to objects of class type. Judicious use of operator overloading can make C++ programs easier to write and easier to read.

Task

This assignment will provide practice with the design and implementation of an ADT and give you practice in C++ concepts such as classes, objects, constructors, operator overloading, friend functions. The task is to implement class `Point` to represent points in a two-dimensional Cartesian coordinate system. You should use the partially completed interface file `point.hpp` and implementation file `point.cpp` to declare and define 2 constructors, 8 member operator overloads, and 15 non-member, non-`friend` functions. Notice there are 0 friend functions.

You'll need a default constructor and a constructor to initialize the `Point` object with x and y coordinates. Since there are no pointers, copy constructor, copy assignment operator, and destructor functions will be implicitly defaulted by the compiler or you could explicitly `default` these functions. You must support the following use cases involving the definition, initialization, and assignment of `Point` objects:

```
Point p0;           // default construction; p0 is (0, 0)
Point p1 = Point{3, 4}; // p1 has coordinates (3, 4)
Point p2 = {3, 4}; // p2 has coordinates (3, 4)
Point p3 {3, 4};    // p3 has coordinates (3, 4)
Point p4(3, 4);     // p4 has coordinates (3, 4)
Point p5 = Point(3, 4); // p5 has coordinates (3, 4)
Point p6(p1);       // copy construction; p6 is (3, 4)
Point p7 = p1;      // copy construction: p7 is (3, 4)
Point p8 {p1};      // copy construction: p8 is (3, 4)
p9 = p4;            // copy assignment; p9 is (3, 4)
Point p10(3);       // error
Point p11 = 3;      // error
```

An important objective of class design is to properly encapsulate the class with a small number of versatile member functions. Clients can then use this minimal yet versatile interface to create new operations that will make their code involving `Point` objects intuitive to code and understand. The 8 member operator overloads consist of:

Operator	Functionality	Description
`[]`	Accessors	2 overloads that take index and return corresponding coordinate.
`+=`	Translation	2 overloads that add two `Point`s or `Point` and `double`. Same behavior as built-in types.
`++`	Prefix/Postfix increment	Increment `Point`'s coordinates. Same behavior as built-in types.
`--`	Prefix/Postfix decrement	Decrement `Point`'s coordinates. Same behavior as built-in types.

The 15 non-member operator overloads consist of:

Operator	Functionality	Description
`%`	Rotation	Return `Point` obtained by rotating `Point` by `double` degrees.
`/`	Distance	Return distance between two `Point`s.
`+`/`-`	Translation	6 functions returning `Point` obtained by adding/subtracting two `Point`s or `Point` and `double` or `double` and `Point` operands.
`-`	Negation	Return `Point` obtained by negating operand's coordinates.
`^`	Midpoint	Return `Point` midway between two `Point`s.
`*`	Dot product	Return `double` representing dot or scalar product of two `Point` operands.
`*`	Scale	2 functions that return `Point` obtained by scaling `Point` and `double` operands.
`<<`	Output	Writes to output stream coordinates of `Point` in format `(x, y)`.
`>>`	Input	Reads two `double`s as `Point` from input stream.

To help you understand the required functionality that the class must implement, a driver program is available that demonstrates the usage of objects of type `Point` by a client. If you need further information, you can consult this faq.

Submission Details

Please read the following details carefully and adhere to all requirements to avoid unnecessary deductions.

Interface and implementation files

You will be submitting both interface file `point.hpp` and implementation file `point.cpp`.

Compiling, executing, and testing

Download `point-driver.cpp`, an input file `input.txt` [to test your overload function for `operator>>`], and a correct output file `output.txt` for unit tests in the driver. After building executable `point.out`, you'll test the program like this:

```
$ ./point.out < input.txt > your-output.txt
$ diff -y --strip-trailing-cr --suppress-common-lines your-output.txt output.txt
```

You should now be familiar with `make` and *makefiles*. Refactor a *makefile* from an earlier lab or assignment and then follow the steps described in previous labs to build and test your program.

There is no need to use `new` nor `delete` operators in this assignment and therefore debugging with Valgrind is not necessary.

File-level and function-level documentation

Every source and header file *must* begin with a *file-level* documentation block. This module will use Doxygen to tag source and header files for generating html-based documentation. In addition, every function that you declare and define and submit for assessment must contain *function-level documentation*. This documentation should consist of a description of the function, the inputs, and return value.

Submission and automatic evaluation

1. In the course web page, click on the appropriate submission page to submit the necessary files.
2. Please read the following rubrics to maximize your grade. Your submission will receive:
   - F grade if your submission doesn't compile with the full suite of `g++` options.
   - F grade if your submission doesn't link to create an executable.
   - Your implementation's output doesn't match correct output of the grader (you can see the inputs and outputs of the auto grader's tests). The auto grader will assign 50% of the grade based on the input and output files given to you. The remaining 50% of the grade will be awarded based on the additional tests implemented by the auto grader.
   - The auto grade will provide a proportional grade based on how many incorrect results were generated by your submission. $A+$ grade if your output matches correct output of auto grader.
   - A deduction of one letter grade for each missing documentation block in your submissions. Each source file must have **one** file-level documentation block and a function-level documentation block for each defined function. A teaching assistant will physically read submitted source files to ensure that these documentation blocks are authored correctly. Each missing or incomplete or copy-pasted (with irrelevant information from some previous assessment) block will result in a deduction of a letter grade. For example, if the automatic grader gave your submission an $A+$ grade and one documentation block is missing, your grade will be later reduced from $A+$ to $B+$. Another example: if the automatic grade gave your submission a C grade and the two documentation blocks are missing, your grade will be later reduced from C to F.

FAQ

1. Define your own driver to test each individual function. My driver will work only if you've implemented the complete interface and therefore it should be used only after you've completed the functions and want a final check before submission.
2. The `operator[]` overloads will do the work of both an accessor and a mutator. Therefore, there's no need to add accessors and mutators with names `GetX`, `GetY`, `SetX`, and `SetY`.
3. The following code fragment illustrates 3 overloads of operator `-` and 2 overloads of `+`:

```
Point pt1 {3, 4}, pt2 {0, 0}, pt3, pt4;

pt3 = -pt1;            // results in pt3 with coordinates (-3, -4)
double d = pt1 - pt2; // results in a double 5.0
pt3 = pt1 - 2;    // results in pt3 with coordinates (1, 2)
pt4 = pt1 + -2;   // overload of operator + results in pt4 (1, 2)
                  // this overload exists to make sure pt3 and pt4
                  // evaluate to same value
pt4 = pt1 + pt2; // results in pt4 with coordinates (3, 4)
pt3 = 2 - pt1;    // ERROR - no operator- exists
```

4. How do you make sure the overloads for prefix increment/decrement and postfix increment/decrement for `Point` have behavior equivalent to built-in types. For example, the following code should not compile!!!

```
Point pt1 {3, 4}, pt2 {0, 0};

// none of the following code should compile!!!
Point pt3   = ++(pt1--);
Point pt4   = --(pt1++);
Point pt5   = ++( pt1 % 45.0 );
Point pt5a  = ( pt1 % 45.0 )++;
Point pt6   = ++( pt1 ^ pt2 );
Point pt6a  = (pt1 ^ pt2 )++;
Point pt7   = ++( pt1 - 5.0 );
Point pt7a  = (pt1 - 5.0 )++;
Point pt8   = ++( -pt1 );
Point pt8a  = (-pt1 )++;
Point pt9   = ++( pt1 + pt2 );
Point pt9a  = ( pt1 + pt2 )++;
Point pt10  = ++( pt1 + 5.0 );
Point pt10a = ( pt1 + 5.0 )++;
Point pt11  = ++( pt1 * 5.0 );
Point pt11a = ( pt1 * 5.0 )++;
Point pt12  = ++( 5.0 + pt1 );
Point pt12a = ( 5.0 + pt1 )++;
Point pt13  = ++( 5.0 * pt1 );
Point pt13a = ( 5.0 * pt1 )++;
```

Why should the above code not compile? Let's begin by analyzing the value of `number` after the statement in line 2?

```cpp
1  int number {0};
2  number++;         // What is being incremented?
3  std::cout << number;
```

If you said `1`, you're correct. This is something you learned a long time ago and even beginner programmers get that correct. OK, suppose you change line 2 to this:

```cpp
(number++)++;     // What is being incremented?
```

Now what is printed? If you said `2`, you'd be wrong and probably quite surprised. The correct answer is that the expression `(number++)++` is illegal and won't compile. Realize that this is also illegal:

```cpp
++(number++);     // What is being incremented?
```

Even if those expressions were valid, `number` would still have a value of `1`, which would only lead to confusion for programmers the world over. Why would `number` still have a value of `1` (assuming the expressions are valid)? To answer that, you have to answer the question: *"What is returned from the postfix increment operator?"*. The answer to that question is: *"a temporary copy of* `number` *"*. A copy is being made so that this kind of statement works as expected:

```cpp
int num1 {0};
int num2 {num1++}; // Returns the "old" value (a copy) of num1
```

After the above code executes, `num1` is `1` and `num2` is `0`. So, what's happening inside the postfix increment operation is that a temporary copy of `num1` is made, then, `num1` is incremented to `1`, and then the temporary copy is used to assign to `num2`. So, if either of these were allowed:

```cpp
(number++)++;     // What is being incremented?
++(number++);     // What is being incremented?
```

the unnamed, compiler-generated, temporary copy of `number` would be incremented, but then that incremented copy is immediately discarded because the programmer has no access to it (it doesn't have a name).

Consider the precedence table for prefix and postfix increment/decrement operators:

Operator	Description	Result	Associativity
`++`	Postfix increment	rvalue	L-R
`--`	Postfix decrement	rvalue	L-R
`++`	Prefix increment	lvalue	R-L
`--`	Prefix decrement	lvalue	R-L

The key is in the **Result** column. The postfix operators return an *rvalue*. That's a fancy name for *unnamed compiler-generated temporary* which are inaccessible to programmers. On the other hand, the prefix increment/decrement operators return an *lvalue*, which represents a named memory location which the programmer can access. That's why this is completely legal and does what you'd expect:

```cpp
int number {0};
++++number;           // Same as ++(++number)
std::cout << number; // Prints value 2
```

Even this is fine:

```cpp
int number {0};
++++++++++number;
std::cout << number; // Prints value 5
```

Since the compiler implements all of the operators for the built-in types (`int`, `long`, `double`, and so on), it enforces this rule and prevents *strange and surprising* behavior that would confuse programmers. However, when you, the programmer, are overloading operators for user-defined types (such as `Point`), the compiler does not enforce this. Which is why you can overload the `'+'` operator to do subtraction. The compiler doesn't care.

To prevent the *strange and surprising* behavior when using the temporary copy returned from the postfix operators, you need to make sure they return constant, that is `const` objects. Finally, this applies to *all* operators that return copies by value, not just the postfix increment and decrement operators.

5. Arguments to C++ standard library `std::sin` and `std::cos` functions must be radians.
6. When implementing the overload for rotation, you're getting `-0.000` instead of just `0.000`. What's the problem and how can you fix it? The problem is with the `double` data type. When using integers, there is only 0; there is no -0. However, with `double`s (or any floating point type), you can have both $+0.0$ and -0.0. Usually, this isn't a problem, but if you round a very small negative number, you will get -0.000. For example, if you have the value `0.00000000000664` and you print it out rounded to 3 decimal places, you'll get `0.000`. But, if you have the value `-0.00000000000664` and you print it out rounded to 3 decimal places, you'll get `-0.000`.

   To fix this, you'll have to check to see how small the value is, and if it is small enough, call it *zero*. For this assignment, you'll say that any value between -0.00001 and $+0.00001$ is zero. This arbitrary value 0.00001 is called an ***epsilon***. You should have code similar to this in your rotation code [for both coordinates]:

```cpp
// if value is between -EPSILON and +EPSILON, make it 0.0
value = (value > -EPSILON && value < EPSILON) ? 0.0 : value;
```

To read up a little more on this notion of epsilon, check out [this page]().